EECE 5644: Introduction to Machine Learning and Pattern Recognition

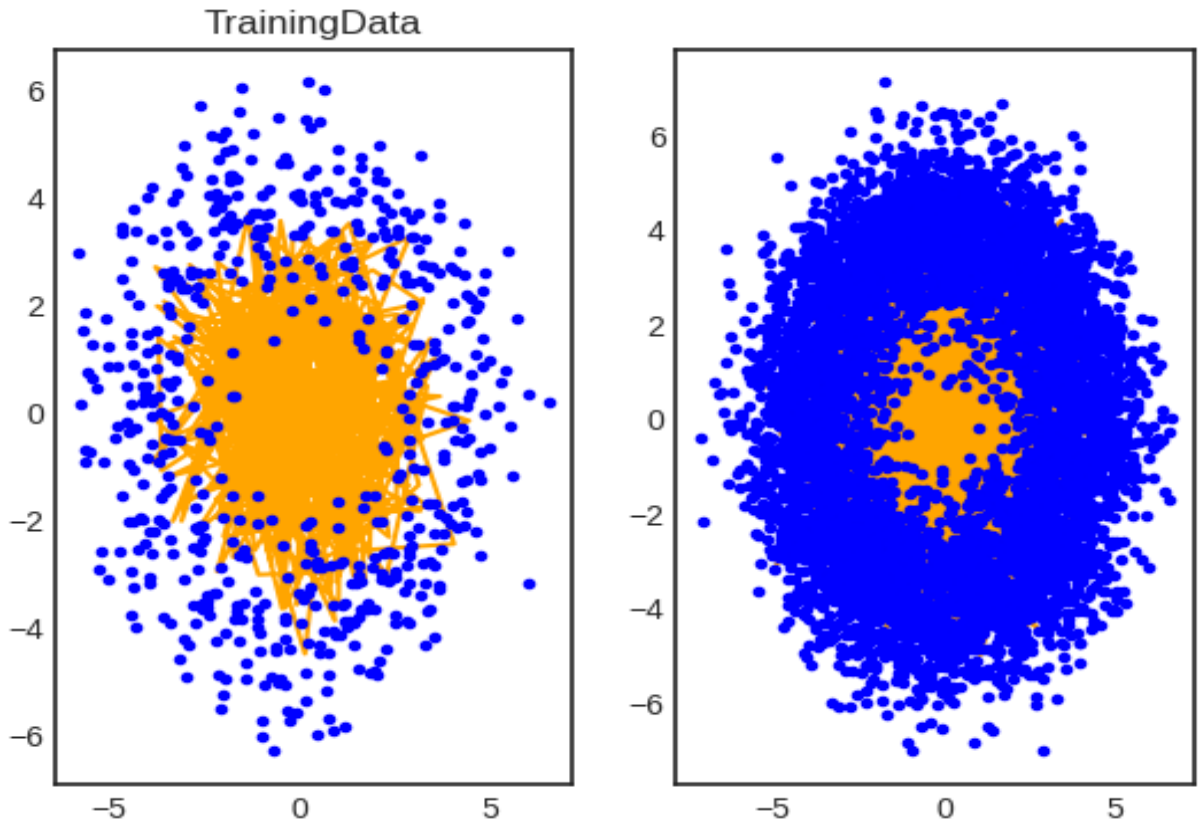
Name: Nikita Vinod Mandal

NUID: 002826995

Assignment 4: Intro to Machine Learning and Pattern Recognition

Question 1:

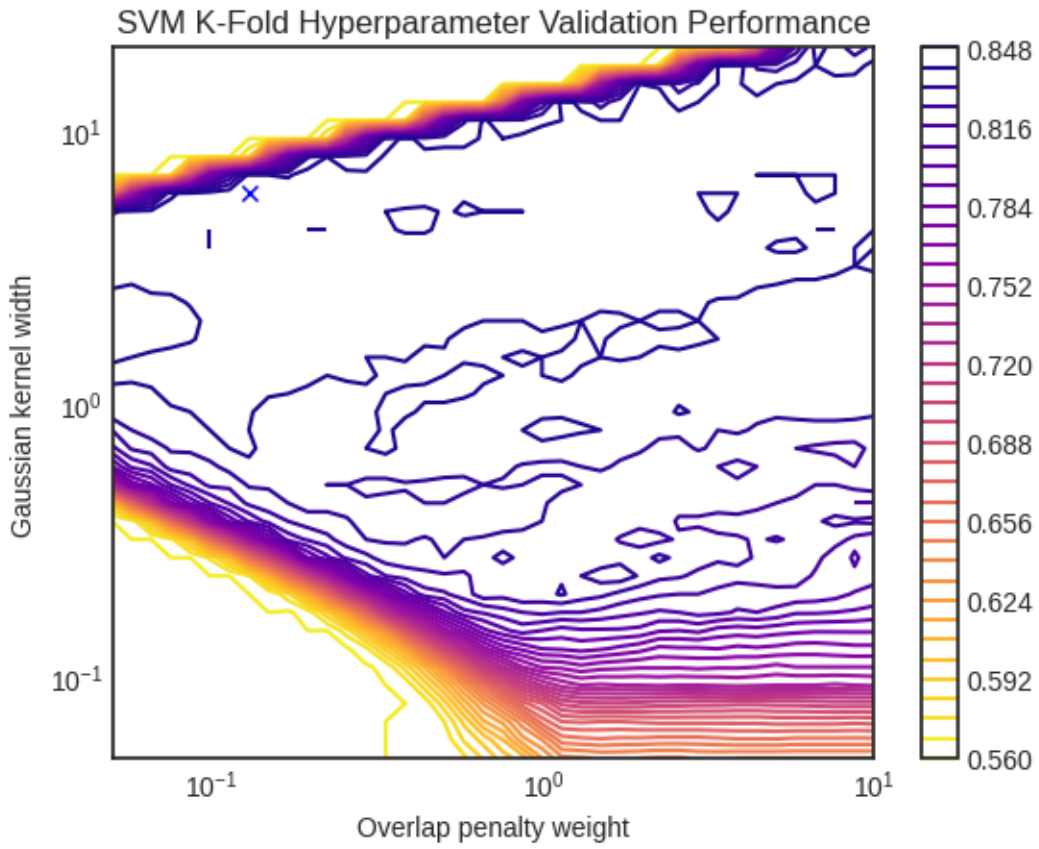
The dataset consists of 1000 independent and identically distributed (iid) samples for training and 10000 iid samples for testing. A Support Vector Machine (SVM) classifier with a Gaussian kernel was trained for data classification. The classes, denoted as -1 and +1, are generated based on $\theta \sim \text{Uniform}[-\pi, \pi]$ and $n \sim N(0, \sigma^2 I)$, where $r_{-1} = 2$, $r_{+1} = 4$, and $\sigma = 1$. The class priors are set to 0.40 for class 0 and $\mathbf{x} = r_l \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} + \mathbf{n}$ 0.60 for class 1. The mean for class 0 is 2, and for class 1, it is 4, with Sigma equal to 1.



Testing and Training Data:

In the process of K-Fold validation for the SVM, 40 values of the overlap penalty weight and Gaussian kernel width were tested within the ranges [0.5, 10] and [0.5, 20], respectively. Ten-fold cross-validation was employed to determine the optimal hyperparameters: the box constraint parameter C and the Gaussian kernel width parameter σ. The selection criterion was the minimum average cross-validation probability of error. After selecting the best hyperparameter values, the entire training set was utilized to train the SVM model, and its performance was evaluated on the test dataset.

The accuracy for a model achieved with each tested combination is shown in the contour plot below.

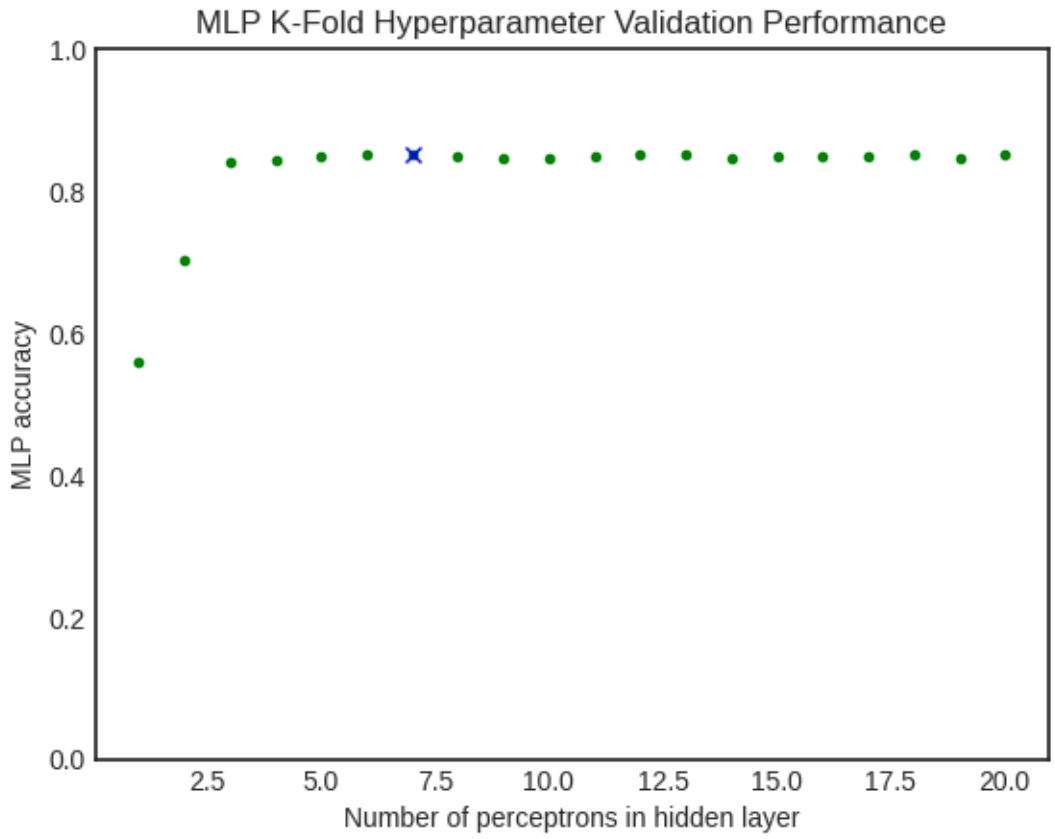


The highest accuracy was attained with an overlap penalty weight of **0.1294122500381292** and a Gaussian kernel width of **5.851592842296028**, denoted by a blue 'x' on the provided plot. **The optimal SVM accuracy during training reached 0.8480000000000001.** This particular combination yielded an average accuracy of **0.848** across the 10 K-Fold validation partitions.

n the plot, accuracy demonstrates increasing flat plateaus on both the lower and upper bounds. When the overlap penalty weight and kernel width are too low, samples are too distant to form meaningful clusters. Conversely, when the overlap penalty weight is low and the kernel width is large, samples become too close, impeding meaningful separation. Striking the right balance between these parameters is crucial for deriving an effective decision boundary, though this boundary may struggle to correctly classify samples within overlapping Gaussian distributions.

The classification boundary exhibits an approximately circular shape, aligning with expectations based on the method of data generation.

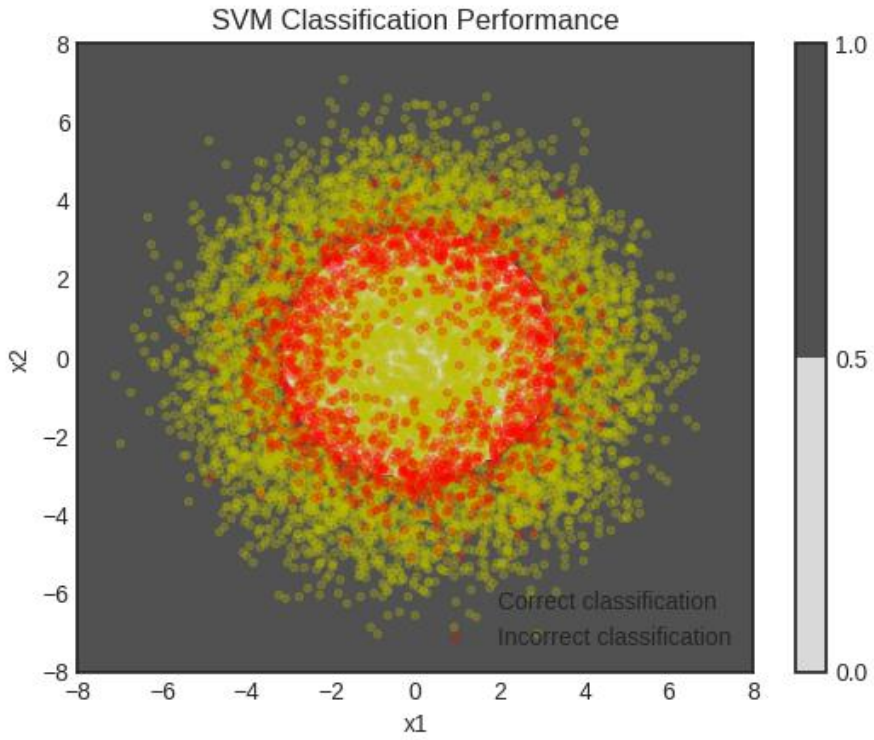
In the context of K-Fold validation for the MLP model, the experiment involved testing up to 7 perceptrons in the hidden layer. The achieved accuracy for each model is illustrated in the plot below.



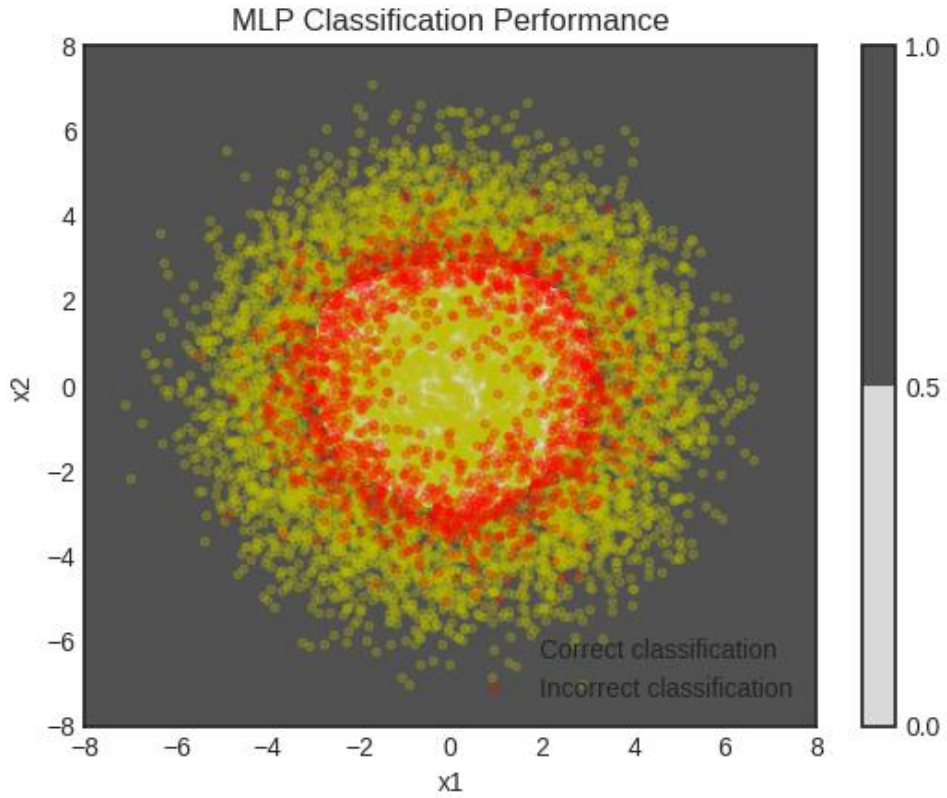
The highest attainable accuracy was observed when utilizing 7 perceptrons in the hidden layer, denoted by a blue 'x' on the provided plot. During training, the optimal MLP configuration achieved an accuracy **0.8509999930858612**. This specific setup resulted in an average accuracy of **0.8499999940395355** across the 10 K-Fold validation partitions. Beyond 7 perceptrons, the accuracy tends to stabilize, indicating that the optimal performance is achieved with the assistance of 7 perceptrons.

Similar to the SVM outcomes, a gradual plateau is evident at the peak accuracy level for the dataset. The K-Fold validation process determined that the optimal model features 7 perceptrons, but performance remains consistent for quantities greater than 2.

The figure below illustrates the outcomes of training the optimal SVM model, as identified through K-Fold validation, on the entire test dataset. Instances of incorrect classification are highlighted in blue, while accurately classified instances are depicted in green.



The below figure was generated by training the optimal MLP model selected by K-Fold validation on the entire test dataset.

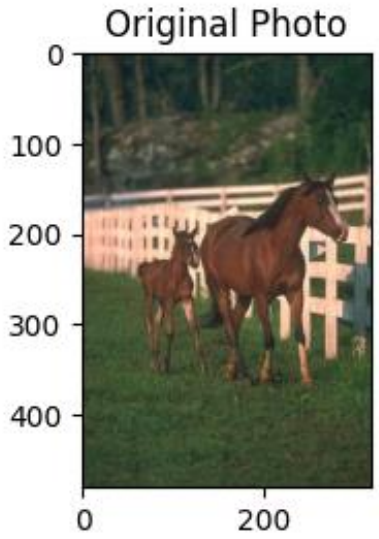


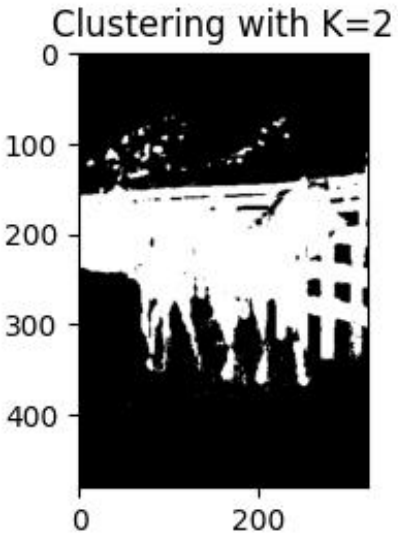
The model mentioned above achieved a fitting accuracy of **0.8375999927520752**. Once again, the classification boundary exhibits an approximately circular shape. In the testing phase, the SVM model achieved an accuracy of **0.8303**, while the MLP model achieved a slightly higher accuracy of **0.8375999927520752**. Despite the MLP classifier achieving a marginally better accuracy, it requires a considerably longer training time.

Visually, the smoother boundary produced by the MLP model closely approaches the ideal circular case compared to the more jagged boundary generated by the SVM model. It's noteworthy that both models operate very close to maximum accuracy, reaching approximately 85% accuracy or a 15% probability of error.

Question 2:

Here is Figure(1) displaying the GMM-based clustering of the original image alongside its 2-component segmentation, taking color into consideration. K-Means Clustering, a form of unsupervised learning, is employed in scenarios where data lacks labels. The primary objective is to identify clusters within the data, with the number of clusters denoted by the variable K.





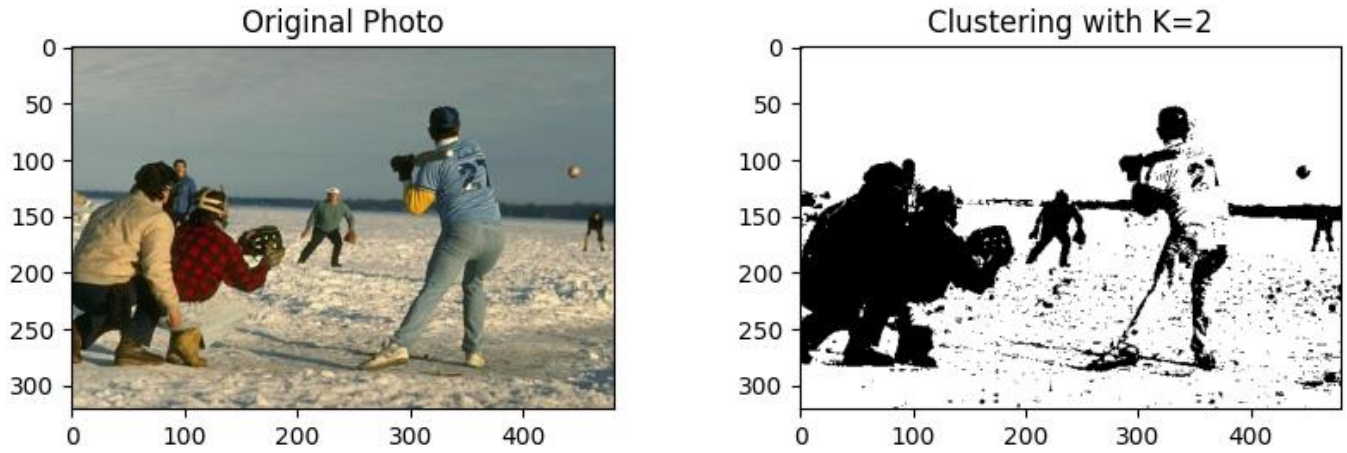
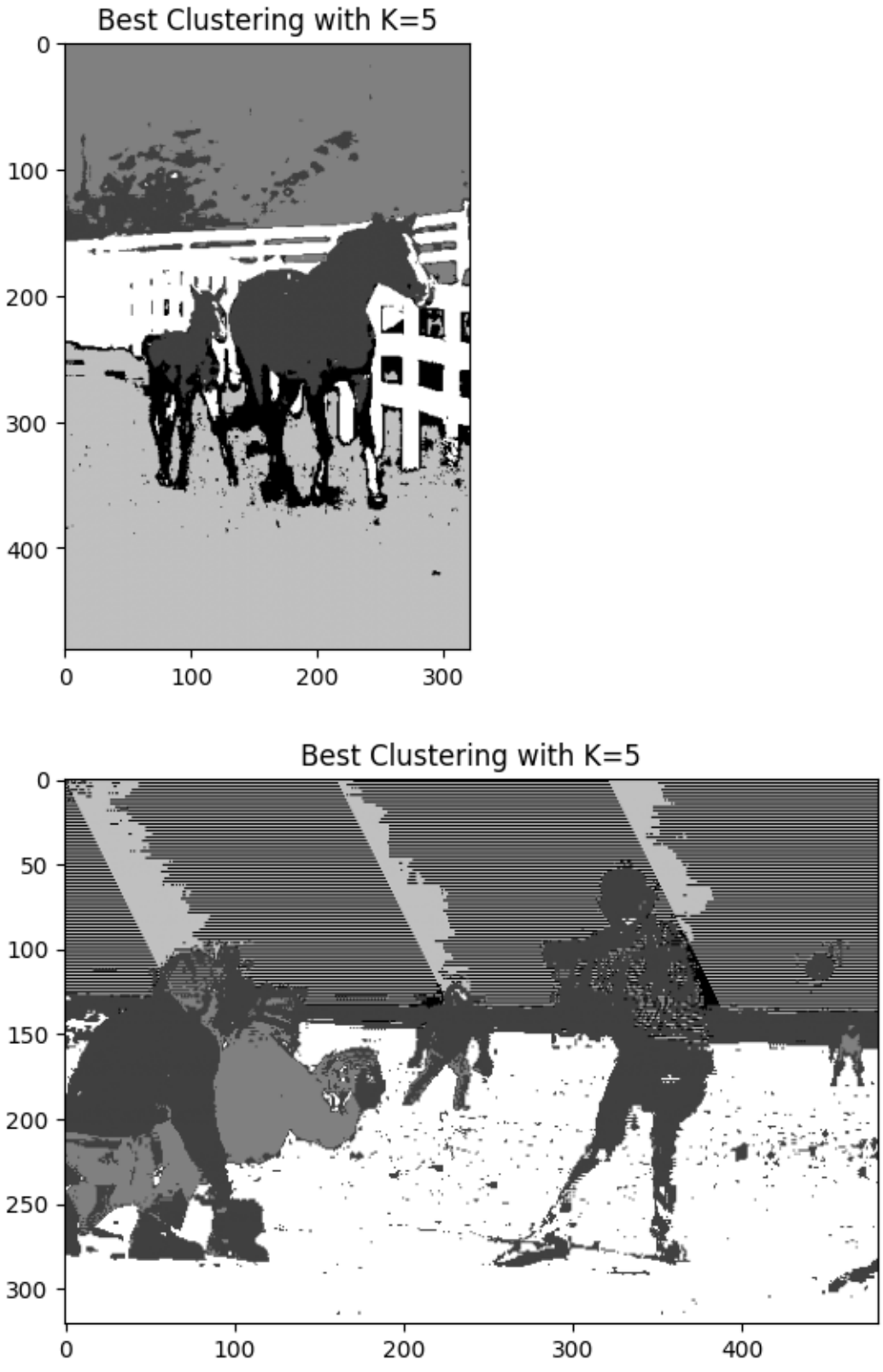


Figure (1)

GMM-based clustering was employed to segment two distinct images— horses and a men playing image.

This Python code employs Gaussian Mixture Models (GMM) for image segmentation. It begins by loading and displaying the original image in grayscale, followed by the creation of a 5-dimensional feature vector for each pixel, incorporating row and column indices, as well as red, green, and blue color values. After normalizing these features, the image is fitted to a 2-component GMM using the **fitgmdist** function.

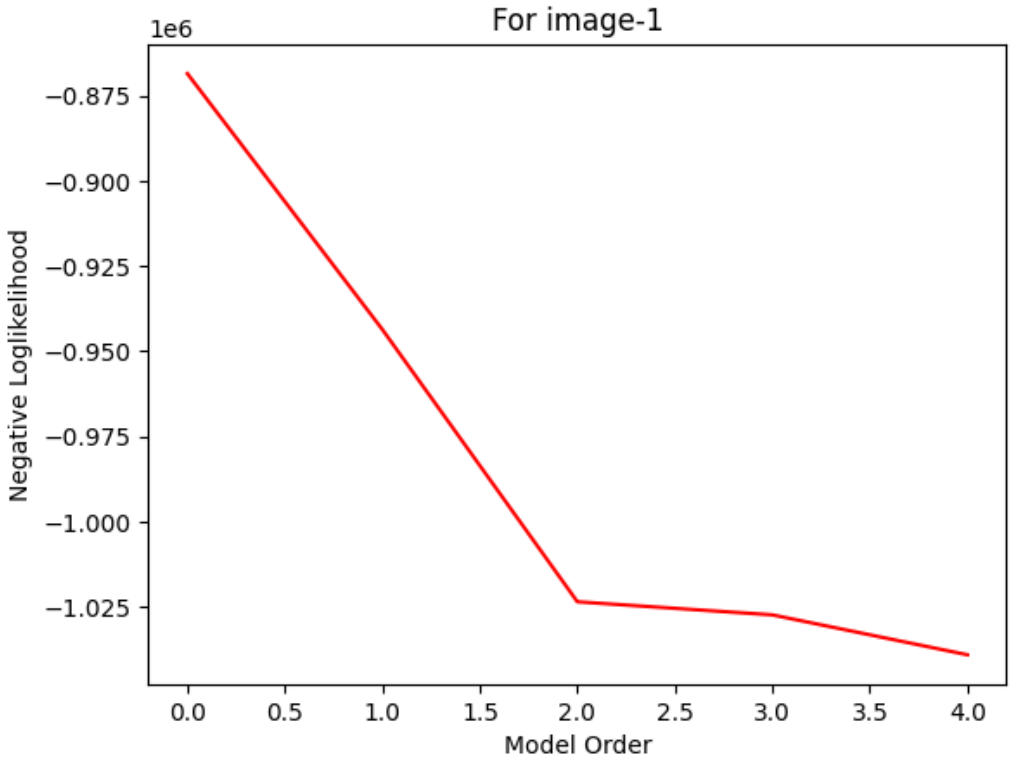
The resulting clustering with K=2 is visualized. The code then performs model selection through likelihood calculation, determining the optimal number of components for GMM. Subsequently, the image is re-segmented with the chosen model order, and the best clustering is presented. The negative log-likelihood is plotted against the model order to assess the model's fit for each image, providing insights into the GMM-based clustering performance.

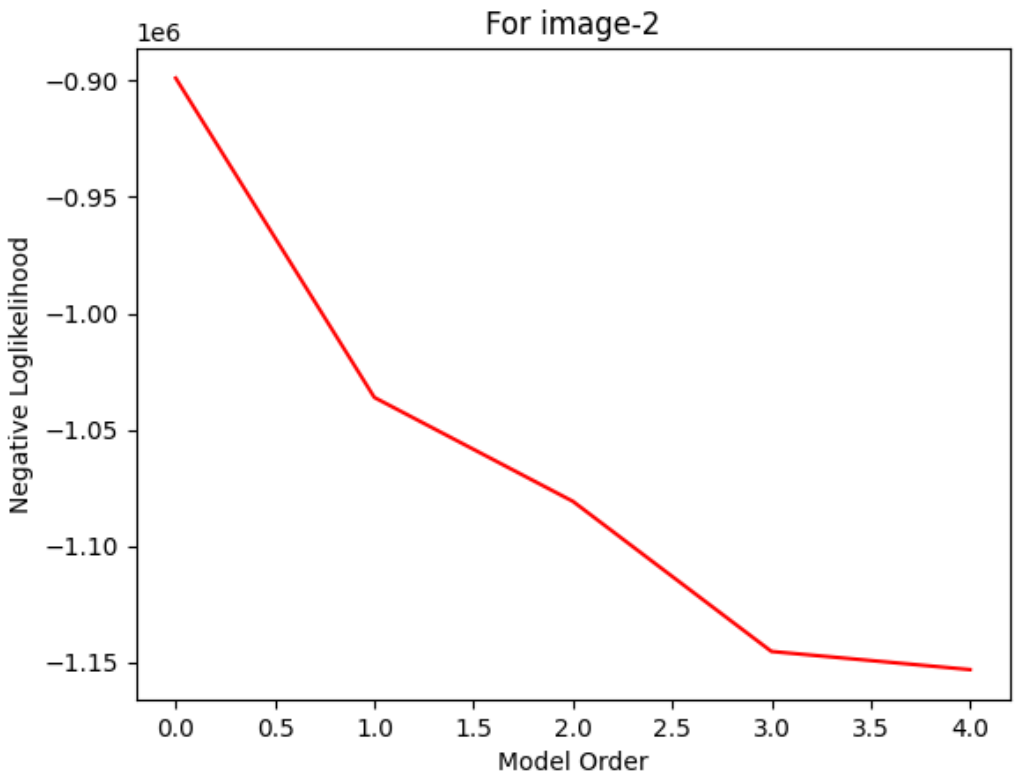

Best Clustering with K=5
0
100
200
300
400
0
100
200
300
Best Clustering with K=5
0
50
100
150
200
250
300
0
100
200
300
400

Figure(2)

The presented Figure 2 showcases the final best-fit Gaussian Mixture Model (GMM) for the given image. In this model, the image underwent fitting, and the cluster function was subsequently applied to assign a cluster number to each pixel. The optimal fit for both the horses and men playing images was achieved with five clusters. This number can be distinct for each cluster concerning the identified properties during the training process.

To achieve the GMM fit, maximum likelihood parameter estimation was employed, coupled with 10-fold cross-validation. The objective during model order selection was to maximize the average validation-log-likelihood. Once the best-fitting GMM was determined, feature vectors were assigned the most likely component label. This assignment was accomplished by evaluating the posterior probabilities of component labels for each feature vector based on the GMM.





In the analysis of each image, a 10-fold cross-validation strategy was employed to fit the image to different model orders, ranging from 1 to 5. The determination of the best-fit model order for each image was based on the average log-likelihood. The hypercube data was subjected to fitting a Gaussian Mixture Model (GMM), and the selection of the optimal number of clusters or classes was guided by the objective function of average-max log-likelihood obtained through 10-fold cross-validation. Notably, for both the images depicting horses and men playing, the algorithm consistently identified 5 clusters as the most suitable. The visualization illustrates the average log-likelihood for each model order and image, where each column number corresponds to a specific model order. As anticipated, the last column (model order 5) exhibits the highest log-likelihood among the tested component values.

Appendix:

Code 1:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import StratifiedKFold
from sklearn.svm import SVC
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

plotData = True
n = 2
Ntrain = 1000
Ntest = 10000
ClassPriors = [0.35, 0.65]
r0 = 2
r1 = 4
sigma = 1

def generate_data(N):
   data_labels = np.random.choice(2, N, replace=True, p=ClassPriors)
   ind0 = np.array((data_labels==0).nonzero())
   ind1 = np.array((data_labels==1).nonzero())
   N0 = np.shape(ind0)[1]
   N1 = np.shape(ind1)[1]
   theta0 = 2*np.pi*np.random.standard_normal(N0)
   theta1 = 2*np.pi*np.random.standard_normal(N1)
   x0 = sigma**2*np.random.standard_normal((N0,n)) + r0 * np.transpose([np.cos(theta0),
   np.sin(theta0)])
   x1 = sigma**2*np.random.standard_normal((N1,n)) + r1 * np.transpose([np.cos(theta1),
   np.sin(theta1)])
   data_features = np.zeros((N, 2))
   np.put_along_axis(data_features, np.transpose(ind0), x0, axis=0)
   np.put_along_axis(data_features, np.transpose(ind1), x1, axis=0)
   return (data_labels, data_features)

def plot_data(TrainingData_labels, TrainingData_features, TestingData_labels,
TestingData_features):
   plt.subplot(1,2,1)
   plt.plot(TrainingData_features[np.array((TrainingData_labels==0).nonzero())][0,:,0],TrainingD
ata_features[np.array((TrainingData_labels==0).nonzero())][0,:,1],'b.')
```

```
  plt.plot(TrainingData_features[np.array((TrainingData_labels==1).nonzero())][0,:,0],
  TrainingData_features[np.array((TrainingData_labels==1).nonzero())][0,:,1],'m.')
  plt.title('TrainingData')
  plt.subplot(1,2,2)
  plt.plot(TestingData_features[np.array((TestingData_labels==0).nonzero())][0,:,0],
TestingData_features[np.array((TestingData_labels==0).nonzero())][0,:,1], 'b.')
  plt.plot(TestingData_features[np.array((TestingData_labels==1).nonzero())][0,:,0],
TestingData_features[np.array((TestingData_labels==1).nonzero())][0,:,1], 'm.')
  plt.show()
# Uses K-Fold cross validation to find the best hyperparameters for an SVM model, and plots
the results

def train_SVM_hyperparams(TrainingData_labels, TrainingData_features):
  hyperparam_candidates = np.meshgrid(np.geomspace(0.05, 10, 40), np.geomspace(0.05, 20,
40))
  hyperparam_performance = np.zeros((np.shape(hyperparam_candidates)[1] *
np.shape(hyperparam_candidates)[2]))
  for (i, hyperparams) in enumerate(np.reshape(np.transpose(hyperparam_candidates), (-1,
2))):
      skf = StratifiedKFold(n_splits=K, shuffle=False)
      total_accuracy = 0
      for(k, (train, test)) in enumerate(skf.split(TrainingData_features, TrainingData_labels)):
          (_, accuracy) = SVM_accuracy(hyperparams, TrainingData_features[train],
TrainingData_labels[train], TrainingData_features[test], TrainingData_labels[test])
          total_accuracy += accuracy
      accuracy = total_accuracy / K
      hyperparam_performance[i] = accuracy
      print(i, accuracy)

  plt.style.use('seaborn-white')
  ax = plt.gca()
  ax.set_xscale('log')
  ax.set_yscale('log')
  max_perf_index = np.argmax(hyperparam_performance)
  max_perf_x1 = max_perf_index % 40
  max_perf_x2 = max_perf_index // 40
  best_overlap_penalty = hyperparam_candidates[0][max_perf_x1][max_perf_x2]
  best_kernel_width = hyperparam_candidates[1][max_perf_x1][max_perf_x2]
  plt.contour(hyperparam_candidates[0], hyperparam_candidates[1],
np.transpose(np.reshape(hyperparam_performance, (40, 40))), cmap='plasma_r', levels=40)
  plt.title("SVM K-Fold Hyperparameter Validation Performance")
  plt.xlabel("Overlap penalty weight")
  plt.ylabel("Gaussian kernel width")
  plt.plot(best_overlap_penalty, best_kernel_width, 'rx')
```

```
    plt.colorbar()
    print("The best SVM accuracy was " + str(hyperparam_performance[max_perf_index]) + ".")
    plt.show()

    return (best_overlap_penalty, best_kernel_width)

def SVM_accuracy(hyperparams, train_features, train_labels, test_features, test_labels):
    (overlap_penalty, kernel_width) = hyperparams
    model = SVC(C=overlap_penalty, kernel='rbf', gamma=1/(2*kernel_width**2))
    model.fit(train_features, train_labels)
    predictions = model.predict(test_features)
    num_correct = len(np.squeeze((predictions == test_labels).nonzero()))
    accuracy = num_correct / len(test_features)
    return (model, accuracy)

def train_MLP_hyperparams(TrainingData_labels, TrainingData_features):
    hyperparam_candidates = list(range(1, 21))
    hyperparam_performance = np.zeros(np.shape(hyperparam_candidates))
    for (i, hyperparams) in enumerate(hyperparam_candidates):
        skf = StratifiedKFold(n_splits=K, shuffle=False)
        total_accuracy = 0
        for(k, (train, test)) in enumerate(skf.split(TrainingData_features, TrainingData_labels)):
            accuracy = max(map(lambda _: MLP_accuracy(hyperparams,
TrainingData_features[train],TrainingData_labels[train], TrainingData_features[test],
TrainingData_labels[test])[1], range(4)))
            total_accuracy += accuracy
        accuracy = total_accuracy / K
        hyperparam_performance[i] = accuracy
        print(i, accuracy)

    plt.style.use('seaborn-white')
    max_perf_index = np.argmax(hyperparam_performance)
    best_num_perceptrons = hyperparam_candidates[max_perf_index]
    plt.plot(hyperparam_candidates, hyperparam_performance, 'b.')
    plt.title("MLP K-Fold Hyperparameter Validation Performance")
    plt.xlabel("Number of perceptrons in hidden layer")
    plt.ylabel("MLP accuracy")
    plt.ylim([0,1])
    plt.plot(hyperparam_candidates[max_perf_index],
hyperparam_performance[max_perf_index], 'rx')
    print("The best MLP accuracy was " + str(hyperparam_performance[max_perf_index]) + ".")
    plt.show()

    return best_num_perceptrons
```

```
def MLP_accuracy(num_perceptrons, train_features, train_labels, test_features, test_labels):
    sgd = SGD(learning_rate=0.05, momentum=0.9)
    model = Sequential()
    model.add(Dense(num_perceptrons, activation='sigmoid', input_dim=2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
    model.fit(train_features, train_labels, epochs=300, batch_size=100, verbose=0)
    (loss, accuracy) = model.evaluate(test_features, test_labels)
    return (model, accuracy)

def plot_trained_model(model_type, model, features, labels):
    predictions = np.squeeze(model.predict(features))
    correct = np.array(np.squeeze((np.round(predictions) == labels).nonzero()))
    incorrect = np.array(np.squeeze((np.round(predictions) != labels).nonzero()))
    plt.plot(features[correct][:,0], features[correct][:,1], 'b.', alpha=0.25)
    plt.plot(features[incorrect][:,0], features[incorrect][:,1], 'r.', alpha=0.25)
    plt.title(model_type + ' Classification Performance')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend(['Correct classification', 'Incorrect classification'])
    gridpoints = np.meshgrid(np.linspace(-8, 8, 128), np.linspace(-8, 8, 128))
    contour_values =
np.transpose(np.reshape(model.predict(np.reshape(np.transpose(gridpoints), (-1, 2))), (128,
128)))
    plt.contourf(gridpoints[0], gridpoints[1], contour_values, levels=1)
    plt.colorbar()
    plt.show()

K = 10
(TRData_labels, TRData_features) = generate_data(Ntrain)
(TestData_labels, TestData_features) = generate_data(Ntest)

if plotData:
    plot_data(TRData_labels, TRData_features, TestData_labels, TestData_features)
    SVM_hyperparams = train_SVM_hyperparams(TRData_labels, TRData_features)
    MLP_hyperparams = train_MLP_hyperparams(TRData_labels, TRData_features)

(overlap_penalty, kernel_width) = SVM_hyperparams
print("The best SVM accuracy was achieved with an overlap penalty weight of " +
str(overlap_penalty) + " and a Gaussian kernel width of " + str(kernel_width) + ".")
print("The best MLP accuracy was achieved with " + str(MLP_hyperparams) + " perceptrons.")
(SVM_model, SVM_performance) = SVM_accuracy(SVM_hyperparams, TRData_features,
TRData_labels, TestData_features, TestData_labels)
```

```
(MLP_model, MLP_performance) = max(map(lambda _: MLP_accuracy(MLP_hyperparams, TRData_features, TRData_labels, TestData_features, TestData_labels), range(5)), key=lambda r: r[1])
print("The test dataset was fit by the SVM model with an accuracy of " + str(SVM_performance) + ".")
print("The test dataset was fit by the MLP model with an accuracy of " + str(MLP_performance) + ".")
plot_trained_model('SVM', SVM_model, TestData_features, TestData_labels)
plot_trained_model('MLP', MLP_model, TestData_features, TestData_labels)
```

Code 2:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from skimage import io, color

def calc_likelihood(x, model, K):
    N = x.shape[1]
    dummy = np.ceil(np.linspace(0, N, K + 1)).astype(int)
    negative_loglikelihood = 0

    ind_partition_limits = np.zeros((K, 2), dtype=int)
    for k in range(K):
        ind_partition_limits[k, :] = [dummy[k] + 1, dummy[k + 1]]

    for k in range(K):
        ind_validate = np.arange(ind_partition_limits[k, 0], ind_partition_limits[k, 1])
        xv = x[:, ind_validate]

        if k == 0:
            ind_train = np.arange(ind_partition_limits[k, 1], N)
        elif k == K - 1:
            ind_train = np.arange(0, ind_partition_limits[k, 0])
        else:
            ind_train = np.arange(ind_partition_limits[k - 1, 1], ind_partition_limits[k + 1, 0])

        xt = x[:, ind_train]

        try:
            gm = GaussianMixture(n_components=model, max_iter=500)
```

```
            gm.fit(xt.T)
            nlogl = -np.sum(gm.score_samples(xv.T))
            negative_loglikelihood += nlogl
        except Exception as e:
            # Handle the exception
            pass

    return negative_loglikelihood

# Initialize
f = ["img2_horse.jpg", "img3.jpg"]
K = 10
M = 5
n = len(f)

# Loop over images
for i in range(n):
    imdata = io.imread(f[i])
    plt.figure(figsize=(10, 6))

    plt.subplot(n, 2, i*2 + 1)
    plt.imshow(imdata, cmap='gray')  # Use 'gray' colormap for grayscale images
    plt.title("Original Photo")

    [R, C, D] = imdata.shape
    N = R * C
    imdata = imdata.astype(float)
    row_indices, col_indices = np.meshgrid(np.arange(1, R + 1), np.arange(1, C + 1))
    features = np.vstack((row_indices.ravel(), col_indices.ravel()))

    for d in range(D):
        imdata_d = imdata[:, :, d]
        features = np.vstack((features, imdata_d.ravel()))

    minf = np.min(features, axis=1)
    maxf = np.max(features, axis=1)
    ranges = maxf - minf
    x = np.diag(1 / ranges) @ (features - np.tile(minf[:, np.newaxis], (1, N)))
    model = 2

    gm = GaussianMixture(n_components=model)
    gm.fit(x.T)
    p = gm.predict(x.T)
    li = p.reshape((R, C))
```

```
    plt.subplot(n, 2, i*2 + 2)
    plt.imshow(li * 255 / model, cmap='gray')  # Use 'gray' colormap for grayscale images
    plt.title(f"Clustering with K={model}")

    ab = np.zeros(M)

    for model in range(1, M + 1):
        ab[model - 1] = calc_likelihood(x, model, K)

    mini = np.argmin(ab)
    gm = GaussianMixture(n_components=mini + 1)
    gm.fit(x.T)
    p = gm.predict(x.T)
    li = p.reshape((R, C))

    plt.figure()
    plt.imshow(li * 255 / (mini + 1), cmap='gray')  # Use 'gray' colormap for grayscale images
    plt.title(f"Best Clustering with K={mini + 1}")

    plt.figure()
    plt.plot(ab, '-r')
    plt.title(f"For image-{i + 1}")
    plt.xlabel("Model Order")
    plt.ylabel("Negative Loglikelihood")

# Show the plots
plt.show()
```

Credits:

1. Prof. Erdogmus Deniz notes.
2. Github